Push-Down Trees: Optimal Self-Adjusting Complete Trees

Chen Avin[1] Kaushik Mondal[1] Stefan Schmid[2]
[1] Ben Gurion University of the Negev, Israel [2] University of Vienna, Austria
avin@cse.bgu.ac.il mondal@post.bgu.ac.il stefan_schmid@univie.ac.at

Abstract

Since Sleator and Tarjan's seminal work on self-adjusting lists, heaps and binary search trees, researchers have been fascinated by dynamic datastructures and the questions related to their performance over time. This paper initiates the study of another classic datastructure, self-adjusting (binary) Complete Trees (CTs): trees which do not provide a simple search mechanism but allow to efficiently access items given a global map. Our problem finds applications, e.g., in the context of warehouse optimization or self-adjusting communication networks which can adapt to the demand they serve.

We show that self-adjusting complete trees assume an interesting position between the complexity of self-adjusting (unordered) lists and binary search trees. In particular, we observe that in contrast to lists, a simple move-to-front strategy alone is insufficient to achieve a constant competitive ratio. Rather, and similarly to binary search trees, an additional (efficient) tree update rule is needed. Intriguingly, while it is unknown whether the *working set* is a lower bound for binary search trees, we show that this holds in our model. So while finding an update rule is still an open problem for binary search trees, this paper shows that there exists a simple, random update rule for complete trees.

Our main result is a dynamically optimal (i.e., constant competitive) self-adjusting CT called Push-Down Tree, on expectation against an oblivious adversary. At the heart of our approach lies a distributed algorithm called Random-Push: this algorithm approximates a natural notion of Most Recently Used (MRU) tree (essentially an *approximate* working set), by first performing move-to-front, but then pushing less recently accessed items down the tree using a random walk.

1 Introduction

Self-adjusting datastructures, introduced over 30 years ago by Sleator and Tarjan [21], are an important and intensively studied concept in the algorithm research community. A self-adjusting datastructure has the appealing property that it can optimize itself to the workload, leveraging temporal locality, but without knowing the future. Ideally, self-adjusting datastructures should store items which will be accessed (frequently) *in the future*, in a way that they can be accessed quickly, while also accounting for reconfiguration costs.

In this work, we study a novel flavor of such self-adjusting structures where lookup is supported by a *map* (i.e., the structure does not have to be searchable). Let us consider the following motivating example. Consider a simplified packing & shipping scenario arising, e.g., in a book warehouse with a single worker, Bob. Bob's desk is placed in the front of the warehouse where he receives a continuous sequence σ of requests for items (here books) that need to be packed and shipped, one item at a time. When a new request for an item v arrives, Bob uses a *map* (e.g., drawn on a whiteboard) of the warehouse to look up the book's rack v; he then walks there to collect the book for shipping (or he sends a robot). Bob's goal is to minimize the total walking distance (e.g., saving time, energy). Fortunately, the warehouse infrastructure supports dynamic placements of items.

The warehouse infrastructure contains a set of *nodes*, each of which can host a single bookrack. Nodes are connected by a network of tracks (railways) on which racks can be shifted (dragged) from one node location to another (along with the books they store). Upon receiving an item request, Bob first needs to *access* the item, but then he can also *rearrange* (i.e., self-adjust) rack locations as he wishes, taking into account the distance he needs to travel to do so. Subsequently, he can update the warehouse map (at no cost) for the next request. We ask: How can Bob optimize his actions, without knowing a priori the items request sequence σ? Can Bob exploit *temporal locality* in the sequence σ if it exist?

We present a formal abstraction for this problem as a self-adjusting datastructure later, but few observations are easy to make. If the network of tracks is an (infinite) line then this problem can be formulated as a *list update problem* [21] for which it is well-known that a simple Move-To-Front (MTF) rule results in a *"dynamically optimal"* self-adjusting list: an online algorithm On, which serves each access without any knowledge of future accesses, is said to be *dynamically optimal* if it executes σ in time $O(\text{cost}(\text{Opt}))$, where cost(Opt) is the cost of an optimal offline algorithm Opt which *knows* σ a priori. We also note that if the network is a line (i.e., a list), then the warehouse map is of no use: Bob can just walk along the single route until he finds the correct bookrack.

In this paper we consider the case where the network of nodes (racks) is a (binary) Complete Tree (CT) and the distance is measured as the number of hops on the shortest path between nodes. A tree network is a generalization of the list, and as we mentioned, maintaining temporal locality in a list is simple: the move-to-front algorithm fulfills the *most-recently used* property, i.e., the i^{th} furthest away item from the front is the i^{th} most recently used item. In the list, this property is enough to guarantee optimality [21], and is essentially a working set property: the working set number $t_i(x)$ of an item x at time i is the number of distinct elements accessed since the last access of x prior to time i, including x; a data structure has the working-set property if the amortized cost of access x is $O(\log t_i(x))$. Naturally, we wonder whether the *most-recently used* property is enough to guarantee optimality in binary trees. The answer turns out to be non-trivial. Our first contribution is to show that if we count only *access* cost (ignoring any rearrangement cost), the answer is affirmative: the most-recently used tree is what is called *access optimal*. But securing this property, i.e., maintaining the most-recently used items close to the root in the tree, introduces a new challenge: how to achieve this *at low cost*? In particular, assuming that *swapping* the locations of items comes at a *unit cost*, can the property be maintained at cost proportional to the *access* cost? As we show, *strictly* enforcing the most-recently used property in a tree is too costly to achieve optimality. But, when turning to an *approximate* most-recently used property, we are able to show two important properties: *i)* such an approximation is good enough to guarantee access optimality; and *ii)* it can be maintained in expectation using a random walk based algorithm.

1.1 Our Contributions

Our main contribution is a dynamically optimal, i.e., constant competitive (on expectation) self-adjusting (binary) Complete Tree (CT) called Push-Down Tree.

Theorem 1 Push-Down Tree *is dynamically optimal on expectation.*

In particular, we show that for CTs, an efficient tree update rule exists: an open problem in the context of self-adjusting binary search trees. At the heart of our technical contribution lies the study and approximation of a natural notion of *Most-Recently Used (MRU) tree*, which can be seen as a *working set* property: An MRU tree T^*, at any given time t, has the property that for any item pair v_1, v_2, if v_1 has been accessed more recently than v_2, the depth of v_1 in T^* is no larger than the depth of v_2 in T^*.

Since maintaining *strict* MRU properties is costly (in terms of adjustment costs), we define the *approximate* MRU tree (resp. *approximate* working set). In a β-*approximate* MRU tree T, short $MRU(\beta)$, it holds that for any item which is at depth d in T^*, it must be within depth $d + \beta$ in T for any d. We investigate different algorithms to ensure the approximate MRU property. In particular, we introduce a distributed algorithm RANDOM-PUSH which provides a constant MRU approximation: it promotes accessed items to the tree root immediately. However, unlike existing self-adjusting binary search tree algorithms, in order to make space for the new item at the root, it uses a simple strategy, based on *random walks*, to push items down the tree in a balanced manner, preserving most recently used items close to the root.

To the best of our knowledge, the design of self-adjusting CTs which do not provide a simple search operation, has not been investigated in the literature so far.

1.2 Novelty and Relationship to Prior Work

Role of the Map. It is important to discuss the role of the *map* in our model. It our setting, the map is global and centralized, and allows us to trivially access a node (or item) at distance k from the front at a cost k. In our example, Bob does not need to *search* for an item in the network; rather, there is an instruction of how to reach the item, before he leaves its desk. This is in striking contrast to self-adjusting binary *search* trees, where no global map is needed and Bob can reach an item at distance k, at a cost k, by greedily searching it in the tree *after* leaving his desk. Interestingly, since Sleator and Tarjan introduced self-adjusting binary search trees over 30 years ago [22], the quest for a constant competitive ratio, i.e., *"dynamically optimal"* algorithm, is still a major open problem. Nevertheless, there are self-adjusting binary search trees that are known to be *access optimal* [6], but their rearrangement cost it too high.

This positions our model, self-adjusting binary trees with a map, in an intriguing new location on the spectrum between dynamic lists and binary search trees. The novelty in our model is that searching items is done centrally (at no or negligible cost), but access and rearranging is distributed and comes at a cost.

One practical motivation (besides Bob's warehouse) to study this model stems from its applications in *distributed and networked systems*, which are becoming increasingly flexible. For example, emerging optical technologies allow to adjust the *physical* topology of a datacenter or wide-area network, in an online manner (e.g., [17]). In particular, a tree may describe a reconfigurable network topology where a communication *source* arranges its communication partners in the form of a bounded degree tree [5, 19]. Accordingly, we in this paper are interested in *distributed algorithms* for self-adjusting trees in which items (e.g., communication partners) can be swapped between neighboring nodes, at unit cost. This poses two key challenges: (1) we need to strike a balance between the *benefits* of adjustments (e.g., reduced communication costs) and their *costs* (the reconfigurations); and (2) provide an efficient routing algorithm on the unordered tree. We discuss this more in Section 5.

Dynamic List Update: Linked List (LL). The dynamically optimal linked list datastructure is a seminal [21] result in the area: algorithms such as Move-To-Front (MTF), which moves each accessed element to the front of the list, are known to be 2-competitive, which is optimal [1, 4, 21]. We note that the Move-To-Front algorithm results in the Most Recently Used property where items that were more recently used are closer to the head of the list. The best known competitive ratio for randomized algorithms for LLs is 1.6, which almost matches the randomized lower bound of 1.5 [3, 23].

Binary Search Tree (BST). Self-adjusting BSTs were the first efficient datastructures studied in the literature. In contrast to CTs, self-adjustments in BSTs are based on *rotations* (which

are assumed to have unit cost). While BSTs have the working set property, we are missing a matching lower bound: the *Dynamic Optimality Conjecture*, the question whether splay trees [22] are dynamically optimal, continues to puzzle researchers even in the randomized case [2]. On the positive side, over the last years, many deep insights into the properties of self-adjusting BSTs have been obtained [10], including improved (but non-constant) competitive ratios [7], regarding weaker properties such as working sets, static, dynamic, lazy, and weighted, fingers, regarding pattern-avoidance [9], and so on. It is also known (under the name *dynamic search-optimality*) that if the online algorithm is allowed to make rotations for free after each request, dynamic optimality can be achieved [6]. Known lower bounds are by Wilber [24], by Demaine et al. [13]'s interleaves bound (a variation), and by Derryberry et al. [14] (based on graphical interpretations). It is not known today whether any of these lower bounds is tight.

Unordered Tree (UT). We are not the first to consider *unordered* trees and it is known that existing lower bounds for (offline) algorithms on BSTs also apply to UTs that use rotations: Wilber's theorem can be generalized [15]. However, it is also known that this correspondance between ordered and unordered trees no longer holds under weaker measures such as *key independent processing costs* and in particular *Iacono's measure* [16]: the expected cost of the sequence which results from a random assignment of keys from the search tree to the items specified in an access request sequence. Iacono's work is also one example of prior work which shows that for specific scenarios, working set and dynamic optimality properties are equivalent. Regarding the current work, we note that the reconfiguration operations in UTs are more powerful than the swapping operations considered in our paper: a rotation allows to move entire subtrees at unit costs, while the corresponding cost in CTs is linear in the subtree size. We also note that in our model, we cannot move freely between levels, but moves can only occur between parent and child. In contrast to UTs, CTs are bound to be balanced.

Skip List (SL) and B-Trees (BT). Intriguingly, althrough SLs and BSTs can be transformed to each other [12], Bose et al. [8] were able to prove dynamic optimality for (a restricted kind of) SLs as well as BTs. Similarly to our paper, the authors rely on a connection between dynamic optimality and working set: they show that the working set property is sufficient for their restricted SLs (for BSTs, it is known that the working set is an upper bound, but it is not known yet whether it is also a lower bound). However, the quest for proving dynamic optimality for general skip lists remains an open problem: two restricted types of models were considered in [8], bounded and weakly bounded. In the bounded model, the adversary can never forward more than B times on a given skip list level, without going down in the search; and in the weakly bounded model, the first i highest levels contain no more than $\sum_{j=0}^{i} B^j$ elements. Optimality only holds for constant B. The weakly bound model, is related to a complete B-ary tree (similar to our complete binary tree), but there is no obvious or direct connection between our result and the weakly bounded optimality. Due to the relationship between SLs and BSTs, a dynamically optimal SL would imply a working set lower bound for BST. Moreover, while both in their model and ours, proving the working set property is key, the problems turn out to be fundamentally different. In contrast to SLs, CTs revolve around *unordered* (and balanced) trees (that do not provide a simple search mechanism), rely on a different reconfiguration operation (i.e., swapping or *pushing* an item to its parent comes at unit cost), and, as we show in this paper, actually provide dynamic optimality for their general form.

Online Paging. More generally, our work is also reminiscent of *distributed heaps* [18] resp. *online paging* models for *hierarchies* of caches [25], which aim to keep high-capacity nodes resp. frequently accessed items close to each other, however, without accounting for the reconfiguration cost over time. Similar to the discussion above, self-adjusting CTs differ from paging models in that in our model, items cannot move arbitrarily and freely between levels (but only between parent and child at unit cost).

1.3 Paper Organization

Section 2 presents our model and Section 3 introduces access-optimal MRU trees. Section 4 describes how to maintain such trees and hence implement self-adjusting trees, and Section 5 discusses applications to networks. We conclude our contribution in Section 6. Some technical details are postponed to the Appendix.

2 Model and Preliminaries

We are given a complete tree T of n *nodes* $S = \{s_1, \ldots, s_n\}$ which store n *items* $V = \{v_1, \ldots, v_n\}$, one item per node. We will denote by $s_1(T)$ the root of the tree T, or s_1 when T is clear from context, and by s_i.left resp. s_i.right the left resp. right child of node s_i. For any $i \in [1, n]$ and any time t, we will denote by $s_i.\text{guest}^{(t)} \in V$ the item mapped to s_i at time t. Similarly, $v_i.\text{host}^{(t)} \in S$ denotes the node hosting item v_i. Note that if $v_i.\text{host}^{(t)} = s_i$ then $s_i.\text{guest}^{(t)} = v_i$.

All access requests to items originate from the root s_1. Access requests occur over time, forming a (finite or infinite) sequence $\sigma = (\sigma^{(1)}, \sigma^{(2)}, \ldots)$, where $\sigma^{(t)} = v_i \in V$ denotes that item v_i is requested, and needs to be accessed at time t. This sequence (henceforth also called the *workload*) is revealed one-by-one to an online algorithm On.

The *depth* of a node s_i is fixed and describes the distance from the root; it is denoted by s_i.dep. The depth of an item v_i *at time* t is denoted by $v_i.\text{dep}^{(t)}$, and is given by the depth of the node to which v_i is mapped at time t. Note that $v_i.\text{dep}^{(t)} = v_i.\text{host.dep}^{(t)}$.

In order to leverage temporal locality and eventually achieve dynamic optimality, we aim to keep recently accessed items close to the root. Accordingly, we are interested in the time of the *most recent use (mru)* (the most recent access) of an item, henceforth denoted by v_i.mru. Both serving the request and adjusting the configuration comes at a cost:

Definition 1 (Cost) *The cost incurred by an algorithm* Alg *to serve a request* $\sigma^{(t)}$ *to access item* v_i *is denoted by* $\text{cost}(\text{Alg}(\sigma^{(t)}))$*, short* $\text{cost}^{(t)}$*. It consists of two parts,* access *cost (*$\text{acc-cost}^{(t)}$*) and* adjustment *cost (*$\text{adj-cost}^{(t)}$*). We define access cost simply as* $\text{acc-cost}^{(t)} = v_i.\text{dep}^{(t)}$ *since* Alg *can maintain a global* map *and access* v_i *via the shortest path. Adjustment cost,* $\text{adj-cost}^{(t)}$*, is the number of total swaps done by items which change location subsequently, where a single swap means changing position of an item with its parent. The total cost, incurred by* Alg *is then*

$$\text{cost}(\text{Alg}(\sigma)) = \sum_t \text{cost}(\text{Alg}(\sigma^{(t)})) = \sum_t \text{cost}^{(t)} = \sum_t (\text{acc-cost}^{(t)} + \text{adj-cost}^{(t)}).$$

Our main objective is to design online algorithms that perform almost as well as optimal offline algorithms (which know σ ahead of time), even in the worst-case. In other words, we want to devise online algorithms which minimize the competitive ratio:

Definition 2 (Competitive Ratio ρ) *We consider the standard definition of (strict) competitive ratio* ρ*, i.e.,*

$$\rho = \max_\sigma \frac{\text{cost}(\text{On})}{\text{cost}(\text{Opt})}$$

where σ *is any input sequence and where* Opt *denotes the optimal offline algorithm.*

If an online algorithm is constant competitive, independently of the problem input, it is called *dynamically optimal*.

Definition 3 (Dynamic Optimality) *An (online) algorithm* On *achieves* dynamic optimality *if it asymptotically matches the offline optimum on every access sequence. In other words, the algorithm* On *is* $O(1)$*-competitive.*

We also consider a weaker form of competitivity (similarly to the notion of *search-optimality* in related work [6]), and say that On is *access-competitive* if we consider only the access cost of On (and ignore any adjustment cost) when comparing it to Opt (which needs to pay both for access and adjustment). For a randomized algorithm, we consider an oblivious online adversary which does not know the random bits of the online algorithm a priori.

3 Access Optimality of MRU Trees

While for *fixed* trees we know that keeping frequent items close to the root is optimal (cf. Appendix), the design of online algorithms for *adjusting* trees is more involved. In particular, it is known that a *Most-Frequently Used* (MFU) policy is not optimal for lists [21]. A natural strategy could be to try and keep items close to the root which have been frequent "recently". However, this raises the question over which time interval to compute the frequencies. Moreover, changing from one MFU tree to another one may entail high adjustment costs. This section introduces a natural *pendant* to the MFU tree for a dynamic setting: the *Most Recently Used (MRU) tree*. Intuitively, the MRU tree tries to keep the "working set" resp. *recently* accessed items close to the root.

Interestingly, we find that maintaining trees in which recently accessed items are close to the root, is the key to the dynamic optimality of our Push-Down Trees. While the move-to-front algorithm, known to be dynamically optimal for self-adjusting lists, naturally provides such a "most recently used" property, generalizing it to the tree is non-trivial.

Accordingly, we present our Push-Down Tree and its underlying online algorithms in two stages: first, we show that any algorithm that maintains an additive approximation of an MRU tree is *access-competitive*; subsequently, we discuss several options to maintain an approximate MRU tree.

At the heart of our approach lies an algorithm to maintain a constant approximation of the MRU tree at any time. In particular, we prove in the following that a constant additive approximation is sufficient to obtain dynamic optimality. On the other hand, also note that the notion of MRU tree itself needs to be understood as an approximation, and MRU trees are not to be confused with optimal trees: when taking into account both access and adjustment costs, moving an item which is accessed only once to the root (as required by the MRU) is not optimal. With this in mind, let us first formally define MRU and MRU-approximate trees, namely MRU(β) tree.

Definition 4 (MRU Tree) *For a given time* t*,* T *is an* MRU *tree if and only if,*

$$\forall v_i, v_j \quad v_i.\text{mru} > v_j.\text{mru} \Rightarrow v_i.\text{dep} \leq v_j.\text{dep} \tag{1}$$

Alternatively we can define an MRU tree using the *rank* of items, which explicitly defines the level in the MRU tree. The rank of an item v_i, i.e., v_i.rank, at time z, is equal to its working set number $t_z(v_i)$. It follows from Definition 4 that a tree is an MRU tree if and only if $v_i.\text{dep} = \lfloor \log v_i.\text{rank} \rfloor$. Therefore, an MRU tree has the working-set property.

The notion of rank helps us understand the structure of an MRU tree. The root of the tree (level zero) will always host an item of rank one. More generally, nodes in level i will host items that have a rank between $(2^i, 2^{i+1} - 1)$. Upon a request of an item, say v_j with rank r, the rank of v_j is updated to one, and only the ranks of items with rank smaller than r are increased, each by 1.

Therefore, the rank of items with rank higher than r do not change and their level (i.e., depth) in the MRU tree remains unchanged (but they may switch location within the same level). Next we define MRU(β) trees for any constant β.

Definition 5 (MRU(β) Tree) *A tree T is called an* MRU(β) *tree if it holds for any item u that, $u.\text{dep}(T) \leq u.\text{dep}(T^*) + \beta$, where T^* is an MRU tree. Or equally, using the rank, $v_i.\text{dep}(T) = \lfloor \log v_i.\text{rank}(T) \rfloor + \beta$.*

Recall that $v_i.\text{rank}(T) = v_i.\text{rank}(T^*)$, since the rank is independent of the item location. Note that, any MRU(0) tree is also an MRU tree.

Definition 6 (MRU (β) algorithm) *An online algorithm* On *is* MRU(β) *if it uses move-to-front and additionally, for each time t, the tree $T^{(t)}$ that* On *maintains, is an* MRU(β) *tree.*

Next we show that MRU(β) algorithms are access-competitive. Recall that the access cost to an item at depth k is k.

Theorem 2 *Any* On *MRU(β) algorithm is $2(1 + \lceil \frac{\beta}{2} \rceil)$ access-competitive.*

The full proof is in the appendix, and we just present the main points here. For simplicity, first assume that On is an MRU(0) algorithm. We employ a potential function argument: Our potential function is based on the difference in the items' locations between On's tree and Opt's tree. From the definition of MRU(0) tree, $s_i.\text{dep} < s_j.\text{dep}$ implies $s_i.\text{guest.mru}(MRU(0)) > s_j.\text{guest.mru}(MRU(0))$ i.e., s_i has been accessed more recently. Accordingly, we define a pair of nodes (s_i, s_j) as *bad* on Opt's tree if $s_i.\text{dep} < s_j.\text{dep}$ but $s_i.\text{guest.mru}(\text{Opt}) < s_j.\text{guest.mru}(\text{Opt})$, i.e., s_i is at a lower level although s_j has been accessed more recently. Note that in an MRU(0) algorithm, none of the pairs is bad since it maintains a perfect MRU tree at all times. Hence bad pairs appear only on Opt's tree. Here after, in this proof, we use s_i.guest resp. s_i.guest.mru to indicate s_i.guest(Opt) resp. s_i.guest.mru(Opt) if not otherwise mentioned. For a given node s_i, let B_i be equal to one plus the number of *bad* pairs (s_i, s_j) for which $s_i.\text{dep} < s_j.\text{dep}$. Define $B = \prod_{i=1}^{n} B_i$.

We define the potential function $\Phi = \log B$. We consider the occurrence of events in the following order. Upon a request, On adjusts its tree, then Opt performs the rearrangements it requires. Opt needs to access an item before it can swap it with one of its neighbors (e.g., to move it closer to the root). The access cost is equal to the depth of the item and each of the swaps costs one unit, and is legal only between neighbors. There can be multiple swaps of items by Opt between two accesses of On, after the first item is accessed.

Now consider the potential at time t (i.e., before On's adjustment for serving request $\sigma^{(t)}$ and Opt's rearrangements between requests t and $t+1$), $\Phi = \Phi^{(t)}$. Moreover, consider the potential after On adjusted its tree, Φ'. The potential change due to On's adjustment is

$$\Delta\Phi_1 = \Phi' - \Phi = \log B' - \log B = \log \frac{B'}{B} \tag{2}$$

We assume that the initial potential is 0 (i.e., no item was accessed). Since the potential is always positive by definition, we can use it to bound the *amortized* cost of On, amortized(On). Consider a request at time t to an item at depth k in the tree of On. The access-cost is $\text{cost}^{(t)}(\text{On}) = k$ and we would like to have the following bound: $\text{amortized}^{(t)}(\text{On}) \leq \text{cost}^{(t)}(\text{On}) + \Delta\Phi$. Assume that the requested item is at node s_r at depth j in Opt's tree, so Opt must pay at least an access cost of j. First we assume that $j < k$.

Let us compute the potential after On updated its MRU tree. For all nodes for which $s_i.\text{dep} < j$, it holds that $B'_i = B_i + 1$: only the access time of the last accessed node, s_r, changed. That is, for

all nodes for which $s_i.\text{dep} \geq j$ (excluding s_r), $B'_i = B_i$. The potential of the accessed node, s_r, will be $B'_r = 1$, since its last access time changed to t. Now:

$$B' = \prod_{i=1}^{n} B'_i = \left(\prod_{s_i.\text{dep}<j} (B_i + 1) \right) \left(\prod_{\substack{s_i.\text{dep}\geq j \\ i \neq r}} B_i \right) B'_r \leq \frac{2^j}{B_r} \prod_{i=1}^{n} B_i = \frac{2^j}{B_r} B \tag{3}$$

The results follows from $\prod_{i=1}^{n}(B_i + 1) \leq 2^n \prod_{i=1}^{n} B_i$ when $B_i \geq 1$, and by multiplying and dividing by B_r. Also recall that $B'_r = 1$.

Now consider the change in potential $\Delta\Phi_1$. Note that $s_r.\text{dep} = j$ so $B_r \geq (2^{k+1}-1)-(2^{j+1}-1) \geq 2^{k-j}$ (recall that $j < k$).

$$\Delta\Phi_1 = \log \frac{B'}{B} \leq \log \frac{2^j B}{B_r B} = \log \frac{2^j}{B_r} \leq \log \frac{2^j}{2^{k-j}} \leq \log 2^{2j-k} = 2j - k \tag{4}$$

Now we compute the potential change due to Opt's rearrangements between accesses. Consider the potential after Opt adjusted its tree, Φ''. Then the potential change due to Opt's adjustment is $\Delta\Phi_2 = \Phi'' - \Phi' = \log B'' - \log B' = \log \frac{B''}{B'}$.

Let Opt access an item z at s_q from level k', raising it to level $k' - 1$ by swapping it with its parent z_1 at s_{q_1}. We denote the t^{th}-ancestor of z by z_t and the node containing it by s_{q_t}. For all nodes with $s_i.\text{dep} = k' - 1$, except s_{q_1}, $B''_i \leq B_i + 1$ holds, as z_1 goes to level k' from $k' - 1$. For s_{q_1}, $B''_{q_1} \leq B'_q + 2^{k'}$ holds as all the items in layer k' may become bad w.r.t. z. All B''_i with $s_i.\text{dep} < k' - 1$ remain unchanged, i.e., $B''_i = B'_i$ Also all B''_i with $s_i.\text{dep} \geq k'$ remains unchanged except B''_q. Since z_1 from level $k' - 1$ becomes $s_q.guest$ at level k', the bad pairs, if any, associated to z_1 from level k' becomes good. Hence, $B''_q \leq B'_{q_1}$. Similarly, if Opt brings z to level $k' - p$ ($s_{q_{k'-p}}$ contain z), by performing p swaps, then we have (see appendix for details),

$$B'' = \prod_{i=1}^{n} B''_i \leq 2^{2k'+1} B' \tag{5}$$

Now we compute the potential change due to Opt's swaps:

$$\Delta\Phi_2 = \log \frac{B''}{B'} \leq \log 2^{2k'+1} = 2k' + 1 \tag{6}$$

OPT may replace several items between two accesses. Let Opt replace w items between the t-th and the $t + 1$-th access. For the r-th replacement, the potential change is less or equal to $(2k'_r + 1)$ where as Opt pays $(k'_r + p_r)$. Putting it all together, we get

$$\text{amortized}^{(t)}(\text{On}) \leq \text{cost}^{(t)}(\text{On}) + \Delta\Phi_1 + \Delta\Phi_2 \leq 2(j + \sum_{r=1}^{w}(k'_r + p_r)) \leq 2\text{cost}^{(t)}(\text{Opt})$$

and finally

$$\text{cost}(\text{On}) = \sum^{t} \text{amortized}^{(t)}(\text{On}) - (\Phi^{(t)} - \Phi^{(0)}) \leq \sum^{t} 2\text{cost}^{(t)}(\text{Opt}) = 2\text{cost}(\text{Opt}) \tag{7}$$

The case $j \geq k$ and $\beta > 0$ are presented in the appendix. We now turn our attention to the problem of efficiently maintaining an approximate MRU tree. To achieve optimality, we need that

the tree adjustment cost will be proportional to the access cost.

4 Self-Adjusting MRU Trees

We are now left with the challenge that we need to design a tree which on the one hand provides a good approximation of MRU to capture temporal locality by providing fast *access* (resp. *routing*) to items, and on the other hand is also adjustable at low cost over time.

Let us now assume that a certain item $\sigma^{(t)} = u$ is accessed at some time t. In order to re-establish the (strict) MRU property, u needs to be promoted to the root. And in fact, as we will see, this fast promotion to the root is also performed by our self-adjusting tree algorithm described later on. This however raises the question of where to move the item currently located at the root, let us call it v. In order to make space for u at the root while preserving locality, we propose to *push down* items from the root, including item v. However, note that simply pushing items down along the path between u and v (as done in lists) will result in a poor performance in the tree. To see this, let us denote the sequence of items along the path from u to v by $P = (u, w_1, w_2, \ldots, w_\ell, v)$, where $\ell = u.\text{dep}$, *before* the adjustment. Now assume that the access sequence σ is as such that it repeatedly cycles through the sequence P, in this order. The resulting cost per request is in the order of $\Theta(\ell)$, i.e., $\Theta(\log n)$ for $\ell = \Theta(\log n)$. However, an algorithm which assigns (and then fixes) the items in P to the top $\log \ell$ levels of the tree, will converge to a cost of only $\Theta(\log \ell) \in O(\log \log n)$ per request: an exponential improvement.

A better approach seems to be to push down the root along paths "in a balanced manner": the root node v should be pushed down (*by swaping* items locally) along a path of items which have not been accessed recently. Note that in order to keep the adjustment cost comparable to the access cost, the push down operation should also be limited to depth $u.\text{dep}$ (before the adjustment). Since the item at depth $u.\text{dep}$ along this balanced path can be different from u, let us call it $w \neq u$, where $w.\text{host}$ is currently occupied. Accordingly, we propose to move w to $u.\text{host}$, noting that we would like to keep all these operations to be performed at cost $O(u.\text{dep})$, i.e., proportional to the original access cost. Figure 2 presents the outline of this type of algorithms. More systematically, we consider the following three intuitive strategies to push down v:

1. Min-Push: Push down, for each row, the *least-recently used* item.
2. Rotate-Push: Every node pushes down in an alternating ("round-robin") manner.
3. Random-Push: The push down occurs along a simple random walk path.

Before discussing the three strategies, we assume a request to item u which is at depth $u.\text{dep} = k$. Let us first consider the Min-Push strategy. The strategy chooses for each depth $i < u.\text{dep}$, $w_i = \arg\min_{v \in V : v.\text{dep} = i} v.\text{mru}$: the *least* recently accessed item from level i. We then push w_i to the host of w_{i+1}. It is not hard to see that this strategy will actually maintain a perfect MRU tree. However, items with least mru in different levels, i.e., $w_i.\text{host}$ and $w_{i+1}.\text{host}$, may not lie on a connected path. So to push w_i to $w_{i+1}.\text{host}$, we may need to travel all the way from $w_i.\text{host}$ to the root and then from the root to $w_{i+1}.\text{host}$, resulting in a cost of i per level. This accumulates a rearrangement cost of $\sum_{i=1}^{k} i > k^2/2$ to push all the items with least mru at each layer up to layer k. This is not proportional to the original access cost k of the requested item and therefore, leads to a non-constant competitive ratio of $\Omega(\log n)$.

Next, we consider the Rotate-Push strategy. This deterministic strategy is very appealing. Its adjustment cost is k, as the access cost. Moreover it can be shown to guarantee an MRU(1) tree for

a wide range of adversaries. This strategy hence seems to be a natural candidate for a dynamically optimal deterministic strategy. However, we are not able to prove this here. In fact, we can show that the ROTATE-PUSH strategy is not able to maintain a constant MRU approximation against every adversary. For example, a clever (but costly) adversary can cause the tree to store items only on the $2\log n$ nodes along the path from the root to the right-most leaf (see Appendix for details). But, in a perfect MRU tree, these $2\log n$ nodes will be hosted at the top $\log\log n$ levels of the tree. Thus, we turn to the third option.

4.1 The Random-Push Strategy

To overcome the problems with the above two strategies, we propose the RANDOM-PUSH strategy. This is a simple random strategy which selects a random path starting at the root by stepping down the tree to depth $k = u.\text{dep}$ (the accessed item), choosing uniformly at random between the two children of each node. This can be seen as a simple random walk in a directed version of the tree, starting from the root of the tree and of length k steps. Clearly, the adjustment cost of RANDOM-PUSH is also k and its actions are independent of any oblivious online adversary.

Theorem 3 RANDOM-PUSH *maintains an MRU*(4) *(Definition 5) tree on expectation, i.e., the expected depth of the item with rank r is less than $\log r + 3 < \lfloor \log r \rfloor + 4$ for any sequence σ and any time t.*

Proof: To analyze RANDOM-PUSH we will define several random variables for an arbitrary σ and time t (so we ignore them in the notation). W.l.o.g., let v be the item with rank i and let $D(i)$ denote the depth of v. First we note that the support of $D(i)$ is the set of integers $\{0, 1, \ldots, i-1\}$. Next we show the following claim.

Lemma 4 *For every $i > j$, we have that $\mathbb{E}[D(i)] > \mathbb{E}[D(j)]$.*

Proof: The result will use Stochastic Domination [20] (see in the appendix): we show that for $i > j$, $D(i)$ is stochastically larger than $D(j)$, denoted by $D(j) \prec D(i)$. It will then follow from Theorem 9 that $\mathbb{E}[D(i)] > \mathbb{E}[D(j)]$. Let u be an item with rank $j < i$, hence, it was requested more recently than v. The dominance follows from the fact that conditioning that v and u first reached the *same* depth (after the last request of u) then their expected progress of depth will be the same. More formally, let D_{uv} be a random variable that denotes the depth when u's depth equals the depth of v for the *first* time (since the last request of u where its depth is set to 0); and -1 if this never happens. Then by the law of total probability, $\mathbb{E}[D(i)] = \mathbb{E}_{D_{uv}}[\mathbb{E}[D(i) \mid D_{uv}]]$ (and similar for $D(j)$). But since the random walk (i.e., push) is independent of the nodes' ranks, we have for $k \geq 0$ that $\mathbb{E}[D(i) \mid D_{uv} = k] \geq \mathbb{E}[D(j) \mid D_{uv} = k]$. But additionally there is the possibility that they will never be at the same depth (after the last request of u) and that v will always have a higher depth, so $\mathbb{E}[D(i) \mid D_{uv} = -1] > \mathbb{E}[D(j) \mid D_{uv} = -1]$, and the claim follows. □

To understand and upper bound $D(i)$, we will use the Markov chain $\mathcal{M}_i$ over the integers $0, 1, 2 \ldots, i-1$ which denote the possible depths in the tree, see Figure 1. For each depth in the tree $j < i-1$, the probability to move to depth $j+1$ is 2^{-j} and the probability to stay at j is $1 - 2^{-j}$, $i-1$, in an absorbing state. This chain captures the idea that the probability of an item at level j to be pushed by a random walk down the tree (to level larger than j) is 2^{-j}. The chain does describe exactly our algorithm and $D(i)$, but we will use it to prove our bound. First, we consider a random walk described exactly by the Markov chain $\mathcal{M}_i$ with an initial state 0. Let $\ell(i, w)$ denote the random variable of the state of a random walk of length w on $\mathcal{M}_i$. Then we can show:

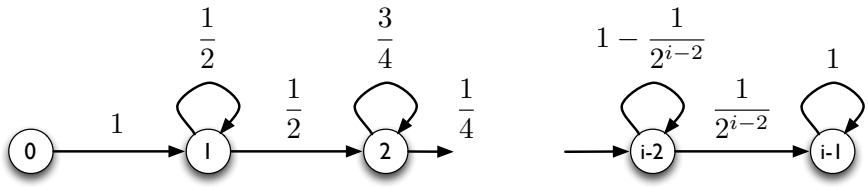

Figure 1: The Markov chain $\mathcal{M}_i$ that is used to prove Theorem 3 and Lemma 5: possible depths for item of rank i in the complete tree.

Lemma 5 *The expected state of $\ell(i,w)$ is such that $\mathbb{E}[\ell(i,w)] < \lceil \log w \rceil + 1$, and $\mathbb{E}[\ell(i,w)]$ is concave in w.*

Proof: Before going to the proof, we have the following corollary.

Corollary 6

$$\sum_{i=0}^{w} \binom{w}{i} (\frac{w-1}{w})^{w-i} (\frac{1}{w})^{i} i = 1 \tag{8}$$

First note that $\mathbb{E}[\ell(i,w)]$ is strictly monotonic in w and can be shown to be concave using the decreasing *rate* of increase: $\mathbb{E}[\ell(i,w_1)] - \mathbb{E}[\ell(i,w_1-\Delta)] \geq \mathbb{E}[\ell(i,w_2)] - \mathbb{E}[\ell(i,w_2-\Delta)]$, for $w_1 < w_2$. To bound $\mathbb{E}[\ell(i,w)]$ we consider a another random walk $\ell'(i,w)$ that starts on state $k = \lceil \log w \rceil$ in a modified chain $\mathcal{M}'_i$. The modified chain $\mathcal{M}'_i$ is identical to $\mathcal{M}_i$ up to state k, but for all states $j > k$ the probability to move to state $j+1$ is $\frac{1}{w} > 2^{-k}$ and the probability to stay at j is $\frac{w-1}{w} < 1 - 2^{-k}$. So clearly $\ell'(i,w)$ makes faster progress than $\ell(i,w)$ from state k onward. The expected progress of $\ell'(i,w)$, starting from state k is now easier to bound and can shown to be: $\mathbb{E}[\ell'(i,w)] = \sum_{i=0}^{w} \binom{w}{i} (\frac{w-1}{w})^{w-i} (\frac{1}{w})^{i} i = 1$. But since $\ell(i,w)$ starts at state 0 we have $\mathbb{E}[\ell(i,w)] < k + 1 = \lceil \log w \rceil + 1$. □

Next we bound the expected number of times that v could be pushed down by a random push. Let W_i be a random variable that denotes the number requests for nodes with higher depth than v, since v's last request until time t.

Lemma 7 *The expected number of requests for items with higher depth than v, since v was last requested, is bounded by $\mathbb{E}[W_i] \leq 2i - 1$.*

Proof: We can divide W_i into two types of requests: $W_i^{>}$ which count the number of requests for items with rank higher than v, and $W_i^{<}$ which count the number of requests for items with lower rank than v (but with higher depth at the time of the request). Then $W_i = W_i^{>} + W_i^{<}$. Clearly $W_i^{>} \leq i$ since every such request increases the rank of v and this happens i times (note that some of these requests may have lower depth than v). $W_i^{<}$ is harder to analyze. How many requests for items are below v in the tree (i.e., have higher depth than v), but also have lower rank than v? (Note that such requests do not increase v's rank, but may increase its depth.) Let u be an item with rank $j < i$, hence u was more recently requested than v. Let X_j denote the number of requests for u (since it was last requested) in which it had a higher depth than v. Then $W_i^{<} = \sum_{n=1}^{i-1} X_n$. We now claim that $\mathbb{E}[X_n] \leq 1$. Assume by contradiction that $\mathbb{E}[X_n] > 1$. But then this implies that the expected depth of u is larger than the expected depth of v, contradicting Lemma 4. Putting it

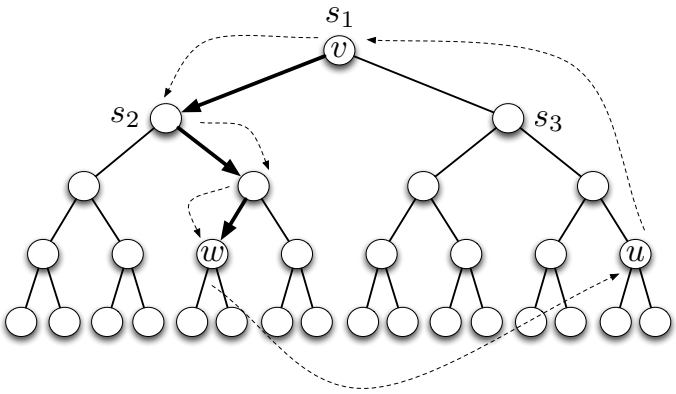

Figure 2: Illustration of the algorithmic framework: After finding the accessed item u, we promote it to the root. The former root item v is then pushed (resp. shifted) down according to a push-down path, up to depth u.dep. The item previously stored at the end of the path, w, is then moved to the former position of u.

all together:

$$\mathbb{E}[W_i] = \mathbb{E}[W_i^{>} + W_i^{<}] \leq \mathbb{E}[i] + \mathbb{E}[\sum_{n=1}^{i-1} X_n] \leq i + \sum_{n=1}^{i-1} \mathbb{E}[X_n] \leq 2i - 1 \tag{9}$$

□

We now have all we need to prove Theorem 3. The proof follows by showing that $\mathbb{E}[D(i)] \leq \log i + 3$. Let $D(i, w)$ be a random variable that denotes the depth of v conditioning that there are w requests of items with higher depth than v, since the last request for v. Note that by the total probability law, we have that $\mathbb{E}[D(i)] = \mathbb{E}_{W_i}[\mathbb{E}[D(i, W_i)]$. Next we claim that $D(i, w) \preceq \ell(i, w)$. This is true since the transition probabilities (to increase the depth) in the Markov chain $\mathcal{M}_i$ are at least as high as in the Markov chain that describes $D(i, w)$. When item v is at some level j, only requests of item u with higher depth $k > j$ can increase v's depth. The probability that a random walk to depth k visits v (and pushes it down) is exactly 2^{-j}. Since $D(i, w) \prec \ell(i, w)$, we have that $\mathbb{E}[D(i, w)] \leq \mathbb{E}[\ell(i, w)]$. Clearly we also have $\mathbb{E}_{W_i}\mathbb{E}[D(i, W_i)] \leq \mathbb{E}_{W_i}\mathbb{E}[\ell(i, W_i)]$. Let $f_i(W_i) = \mathbb{E}[\ell(i, W_i)]$ be a random variable which is a function of the random variable W_i. Recall that $f_i()$ is concave, then by Jensen's inequality [11] and Lemma 7 we get:

$$\begin{aligned} \mathbb{E}[D(i)] &= \mathbb{E}_{W_i}[\mathbb{E}[D(i, W_i)] \leq \mathbb{E}_{W_i}[\mathbb{E}[\ell(i, W_i)]] \\ &= \mathbb{E}_{W_i}[f_i(W_i)] \leq f_i(\mathbb{E}[W_i]) \leq f_i(2i - 1) = \mathbb{E}[\ell(i, 2i - 1)] \leq \lceil \log 2i \rceil + 1 \leq \log i + 3 \end{aligned} \tag{10}$$

□

4.2 Push-Down Trees Are Dynamically Optimal

To conclude, upon access of an item, Push-Down Tree proceeds as follows, see also Algorithm 1 and Figure 2 for an illustration.

1. *Accessed node u:* Upon a request to u, access u along the shortest path from root.
2. *Move-to-root:* Item u is moved to the root, evicting the current at the root, call it v.

3. *Push down items in a balanced manner:* Execute RANDOM-PUSH up to depth u.dep.

4. *Solve overflow:* Move the last item on the push-down path, w, to u.host.

In summary, if $u.\text{dep} = k$, then using RANDOM-PUSH, we have an adjustment cost of k, in addition to the access cost of k. Note that the move of w to u.host does not require any additional swaps and all the participating nodes w.host and u.host have already been accessed, so we assume that no additional cost is required. Even if we assume that this comes at a cost, it is bounded by $2k$ and does not change the main results. Taking the adjustment cost into account, with the help of Theorem 3, we get that PUSH-DOWN TREE is 12-competitive, which is our main result.

Theorem 1 PUSH-DOWN TREE *is dynamically optimal on expectation.*

Proof: According to RANDOM-PUSH, if the t-th requested item has rank r_t, then the access cost is $D(r_t)$ and the push down strategy requires additional $D(r_t)$ swaps (we assume that the last step of moving the last item of the push-down path at the last requested node has constant cost: assuming a cost of $2D(r_t)$ will increase the competitive ratio but will not change the conceptual result). Hence the maintenance cost is $D(r_t)$, which is equal to the access cost. So the expected total cost is two times the access cost on the MRU(4) tree. Formally, using Theorem 2 and Theorem 3:

$$\mathbb{E}\left[\text{cost}(\text{RANDOM-PUSH})\right] = \mathbb{E}\left[\sum_{i=1}^{t} 2D(r_i)\right] = 2\sum_{i=1}^{t}\mathbb{E}\left[D(r_i)\right] \leq 2\sum_{i=1}^{t}(\log(r_i)+3) \tag{11}$$

$$\leq 2\sum_{i=1}^{t}(\lfloor\log(r_i)\rfloor + 4) \leq 2\sum_{i=1}^{t}\text{cost}^{(t)}(\text{MRU}(4))$$

$$\leq 2\cdot\text{cost}(\text{MRU}(4)) = 12\cdot\text{cost}(\text{OPT}) \tag{12}$$

□

5 Application: Self-Adjusting Tree Network

We are particularly interested in distributed applications, and hence, we will now interpret the PUSH-DOWN TREE as a communication network between a *source* and its communication partners. Whenever the source wants to communicate to a partner, it routes to it via the tree. Since nodes in the tree do not know the location of other nodes, a *routing algorithm* is needed: nodes need to know whether to forward a given request to the left or the right child toward the destination.

Note that in a distributed setting, RANDOM-PUSH is performed by iteratively shifting items of neighboring nodes along the downward path. We can observe that both the access and adjustment operations can be performed at cost $O(u.\text{dep})$: all operations are limited to the depth of the accessed node u.dep. To achieve this in a distributed setting, we propose the following. When the root s_1 receives a request for item u, it needs to route to u.host. For this purpose, each node in the tree needs to know, for each request, whether it needs to be forwarded to the left or the right child toward u. A simple way to achieve this in a distributed setting is to employ *source routing*: in principle, a message passed between nodes can include, for each node it passes, a bit indicating which child to forward the message next. This simple solution requires $O(\log n)$ bits in the message header, where n is the number of nodes, and can be used both for accessing the node and for the random push-down path.

The source routing header can be built based on a dynamic global map of the tree that is maintained at the source node. The source node is a direct neighbor of the root of the tree, aware of all requests, and therefore it can maintain the map. Implementing, updating and accessing such a map is memory based and can be done efficiency with negligible cost.

6 Conclusion

This paper presented a dynamically optimal datastructure, called Push-Down Tree, which is based on complete trees. Our algorithms so far are randomized, and the main open theoretical question concerns *deterministic* constructions. Push-Down Trees are also distributed and find interesting applications in emerging self-adjusting communication networks. In this regard, we understand our work as a first step, and the design of fully decentralized and self-adjusting communication networks constitutes our main research vision for the future.

References

[1] Susanne Albers. A competitive analysis of the list update problem with lookahead. *Mathematical Foundations of Computer Science 1994*, pages 199–210, 1994.

[2] Susanne Albers and Marek Karpinski. Randomized splay trees: theoretical and experimental results. *Information Processing Letters*, 81(4):213–221, 2002.

[3] Susanne Albers, Bernhard Von Stengel, and Ralph Werchner. A combined bit and timestamp algorithm for the list update problem. *Information Processing Letters*, 56(3):135–139, 1995.

[4] Susanne Albers and Jeffery Westbrook. Self-organizing data structures. In *Online algorithms*, pages 13–51. Springer, 1998.

[5] Chen Avin, Kaushik Mondal, and Stefan Schmid. Demand-aware network designs of bounded degree. In *Proc. International Symposium on Distributed Computing (DISC)*, 2017.

[6] Avrim Blum, Shuchi Chawla, and Adam Kalai. Static optimality and dynamic search-optimality in lists and trees. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–8, 2002.

[7] Prosenjit Bose, Karim Douïeb, Vida Dujmović, and Rolf Fagerberg. An o (log log n)-competitive binary search tree with optimal worst-case access times. In *Scandinavian Workshop on Algorithm Theory*, pages 38–49. Springer, 2010.

[8] Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and b-trees. In *Proc. 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1106–1114, 2008.

[9] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Pattern-avoiding access in binary search trees. In *Proc. Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 410–423. IEEE, 2015.

[10] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. The landscape of bounds for binary search trees. *arXiv preprint arXiv:1603.04892*, 2016.

[11] T.M. Cover and J. Thomas. *Elements of information theory*. Wiley, 2006.

[12] Brian C. Dean and Zachary H. Jones. Exploring the duality between skip lists and binary search trees. In *Proceedings of the 45th Annual Southeast Regional Conference*, ACM-SE 45, pages 395–399, New York, NY, USA, 2007. ACM.

[13] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu. Dynamic optimality - almost. *SIAM J. Comput.*, 37(1):240–251, 2007.

[14] Jonathan Derryberry, Daniel Dominic Sleator, and Chengwen Chris Wang. *A lower bound framework for binary search trees with rotations*. School of Computer Science, Carnegie Mellon University, 2005.

[15] Michael L Fredman. Generalizing a theorem of wilber on rotations in binary search trees to encompass unordered binary trees. *Algorithmica*, 62(3-4):863–878, 2012.

[16] John Iacono. Key-independent optimality. *Algorithmica*, 42(1):3–10, 2005.

[17] M. Ghobadi et al. Projector: Agile reconfigurable data center interconnect. In *Proc. ACM SIGCOMM*, pages 216–229, 2016.

[18] Christian Scheideler and Stefan Schmid. A distributed and oblivious heap. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 571–582, 2009.

[19] Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker. Splaynet: Towards locally self-adjusting networks. *IEEE/ACM Transactions on Networking (ToN)*, 2016.

[20] Moshe Shaked and J George Shanthikumar. *Stochastic orders*. Springer Science & Business Media, 2007.

[21] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985.

[22] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[23] Boris Teia. A lower bound for randomized list update algorithms. *Information Processing Letters*, 47(1):5–9, 1993.

[24] Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1):56–67, 1989.

[25] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Management of multilevel, multiclient cache hierarchies with application hints. *ACM Transactions on Computer Systems (TOCS)*, 29(2):5, 2011.

Appendix

A Optimal Fixed Trees

The key difference between binary search trees and binary trees is that the latter provides more flexibilities in how items can be arranged on the tree. Accordingly, one may wonder whether more flexibilities will render the optimal data structure design problem algorithmically simpler or harder.

In this section, we consider the static problem variant, and investigate offline algorithms to compute optimal trees for a *fixed* frequency distribution over the items. To this end, we assume that for each item v, we are given a frequency $v.\text{freq}$, where $\sum_{v \in V} v.\text{freq} = 1$.

Definition 7 (Optimal Fixed Tree) *We call a tree* optimal static tree *if it minimizes the expected path length* $\sum_{i \in [1,n]} (v_i.\text{freq} \cdot v_i.\text{dep})$.

Our objective is to design an optimal static tree according to Definition 7. Now, let us define the following notion of *Most Frequently Used (MFU)* tree which keeps items of larger empirical frequencies closer to the root:

Definition 8 (MFU Tree) *A tree in which for every pair of items* $v_i, v_j \in V$*, it holds that if* $v_i.\text{freq} \geq v_j.\text{freq}$ *then* $v_i.\text{dep} \leq v_j.\text{dep}$*, is called* MFU tree.

Observe that MFU trees are not unique but rather, there are many MFU trees. In particular, the positions of items at the same depth can be changed arbitrarily without violating the MFU properties.

Theorem 8 (Optimal Fixed Trees) *Any MFU tree is an optimal fixed tree.*

Proof: Recall that by definition, MFU trees have the property that for all node pairs v_i, v_j: $v_i.\text{freq} > v_j.\text{freq} \Rightarrow v_i.\text{dep} \leq v_j.\text{dep}$. For the sake of contradiction, assume that there is a tree T which achieves the minimum expected path length but for which there exists at least one item pair v_i, v_j which violates our assumption, i.e., it holds that $v_i.\text{freq} > v_j.\text{freq}$ but $v_i.\text{dep} > v_j.\text{dep}$. From this, we can derive a contradiction to the minimum expected path length: by swapping the positions of items v_i and v_j, we obtain a tree T' with an expected path length which is shorter by $\text{cost}(T, \sigma) - \text{cost}(T', \sigma) = v_i.\text{freq} \cdot v_i.\text{dep} + v_j.\text{freq} \cdot v_j.\text{dep} - v_i.\text{freq} \cdot v_j.\text{dep} + v_j.\text{freq} \cdot v_i.\text{dep} > 0$. □

MFU trees can also be constructed very efficiently, e.g., by performing the following *ordered insertion*: we insert the items into the tree T in a top-down, left-to-right manner, in descending order of their frequencies (i.e., item v_i is inserted before item v_j if $v_i.\text{freq} > v_j.\text{freq}$).

B Adversary for Rotate-Push

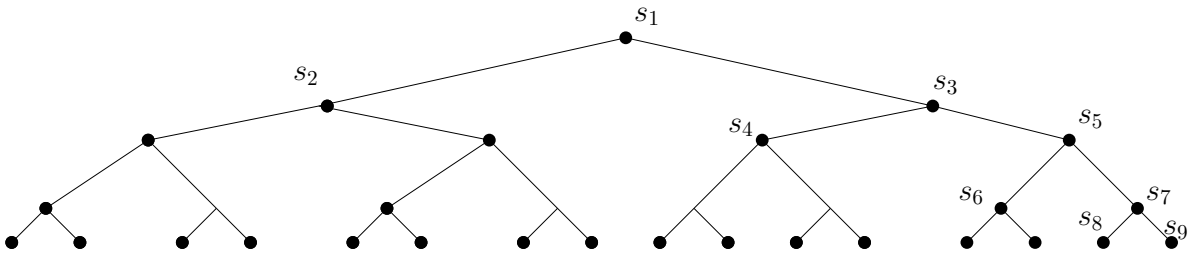

Figure 3: Example for Rotate-Push not maintaining the working set on CT.

We show a complete binary tree of height four in Figure 3 where s_i, $1 \leq i \leq 9$ are some labeled nodes. The root is at level zero and the leaves are at level four. Let us assume that each node pushes to its left child first and then rotates every time. We construct a sequence of requests which prevents Rotate-Push to maintain the working set. Let the requests be from level 4, 1, 2, 1, 3, 1, 4, 1. Consider the first request from level 4. According to the strategy (and Figure 3), the pushdown

path would be $s_1 \to s_3 \to s_5 \to s_7 \to s_9$. Paths for the next requests are respectively $s_1 \to s_2$, $s_1 \to s_3 \to s_4$, $s_1 \to s_2$, $s_1 \to s_3 \to s_5 \to s_6$, $s_1 \to s_2$, $s_1 \to s_3 \to s_5 \to s_7 \to s_8$, and $s_1 \to s_2$ respectively. Now if the adversary continues to generate requests from these nodes maintaining the above sequence, then Rotate-Push is unable to maintain the working set property, which is to keep these requested nine elements within level three in the tree.

Interestingly, although we are not maintaining the working set, we are unable to prove that Rotate-Push does not guarantee a constant approximation of the optimal. This raises the following question: is it necessary to maintain the working set property to guarantee dynamic optimality on a complete binary tree? We are unable to answer this but believe that Rotate-Push maintains dynamic optimality on a complete binary tree. We leave it as a conjecture.

C Deferred Pseudo-Code

Algorithm 1: Upon access to u in Push-Down Tree

1: **route** to $s = u$.host along tree branches (cost: u.dep)
2: let $v = s_1$.guest be the item at the current root
3: **move** u to the root node s_1, setting s_1.guest $= u$
4: employ Random-Push to **shift** down v up to depth s.dep (cost: u.dep)
5: let w be the item at the end of the push-down path, where w.dep $= s$.dep
6: **move** w to s, i.e., setting s.guest $= w$ (cost: u.dep $\times$ 2)

D Deferred Proofs

D.1 Proof of Theorem 2

For simplicity, first assume that On is an MRU(0) algorithm. We employ a potential function argument: Our potential function is based on the difference in the items' locations between On's tree and Opt's tree. From the definition of MRU(0) tree, s_i.dep $< s_j$.dep implies s_i.guest.mru($MRU(0)$) $>$ s_j.guest.mru($MRU(0)$) i.e., s_i has been accessed more recently. Accordingly, we define a pair of nodes (s_i, s_j) as *bad* on Opt's if s_i.dep $< s_j$.dep but s_i.guest.mru(Opt) $< s_j$.guest.mru(Opt), i.e., s_i is at a lower level although s_j has been accessed more recently. Note that in an MRU(0) algorithm, none of the pairs is bad since it maintains a perfect MRU tree at all times. Hence bad pairs appear only on Opt's tree. Here after, in this proof, we use s_i.guest resp. s_i.guest.mru to indicate s_i.guest(Opt) resp. s_i.guest.mru(Opt) if not otherwise mentioned. For a given node s_i, let B_i be equal to one plus the number of *bad* pairs (s_i, s_j) for which s_i.dep $< s_j$.dep. Define $B = \prod_{i=1}^{n} B_i$.

We define the potential function $\Phi = \log B$. We consider the occurrence of events in the following order. Upon a request, On adjusts its tree, then Opt performs the rearrangements it requires. Opt needs to access an item before it can swap it with one of its neighbors (e.g., to move it closer to the root). The access cost is equal to the depth of the item and each of the swaps costs one unit, and is legal only between neighbors. There can be multiple swaps of multiple items by Opt between two accesses of On, after the first item is accessed.

Now consider the potential at time t (i.e., before On's adjustment for serving request $\sigma^{(t)}$ and Opt's rearrangements between requests t and $t+1$), $\Phi = \Phi^{(t)}$. Moreover, consider the potential

after On adjusted its tree, Φ'. Then the potential change due to On's adjustment is

$$\Delta\Phi_1 = \Phi' - \Phi = \log B' - \log B = \log \frac{B'}{B} \tag{13}$$

We assume that the initial potential is 0 (i.e., no item was accessed). Since the potential is always positive by definition, we can use it to bound the *amortized* cost of On, amortized(On). Consider a request at time t to an item at depth k in the tree of On. The access-cost is $\text{cost}^{(t)}(\text{On}) = k$ and we would like to have the following bound: $\text{amortized}^{(t)}(\text{On}) \leq \text{cost}^{(t)}(\text{On}) + \Delta\Phi$. Assume that the requested item is at node s_r at depth j in Opt's tree, so Opt must pay at least an access cost of j. First we assume that $j < k$.

Let us compute the potential after On updated its MRU tree. For all nodes for which $s_i.\text{dep} < j$, it holds that $B'_i = B_i + 1$: only the access time of the last accessed node, s_r, changed. That is, for all nodes for which $s_i.\text{dep} \geq j$ (excluding s_r), $B'_i = B_i$. The potential of the accessed node, s_r, will be $B'_r = 1$, since its last access time changed to t. Now:

$$B' = \prod_{i=1}^{n} B'_i = \left(\prod_{s_i.\text{dep}<j} (B_i + 1) \right) \left(\prod_{\substack{s_i.\text{dep}\geq j \\ i \neq r}} B_i \right) B'_r \tag{14}$$

$$\leq \left(2^j \prod_{s_i.\text{dep}<j} B_i \right) B_r \left(\prod_{\substack{s_i.\text{dep}\geq j \\ i \neq r}} B_i \right) \frac{B'_r}{B_r} = \frac{2^j}{B_r} \prod_{i=1}^{n} B_i = \frac{2^j}{B_r} B \tag{15}$$

The second line results from $\prod_{i=1}^{n}(B_i + 1) \leq 2^n \prod_{i=1}^{n} B_i$ when $B_i \geq 1$, and by multiplying and dividing by B_r. Also recall that $B'_r = 1$.

Now consider the change in potential $\Delta\Phi_1$. Note that $s_r.\text{dep} = j$ so $B_r \geq (2^{k+1}-1)-(2^{j+1}-1) \geq 2^{k-j}$ (recall that $j < k$).

$$\Delta\Phi_1 = \log \frac{B'}{B} = \log \frac{2^j B}{B_r B} = \log \frac{2^j}{B_r} \leq \log \frac{2^j}{2^{k-j}} \leq \log 2^{2j-k} = 2j - k \tag{16}$$

Now we compute the potential change due to Opt's rearrangements between accesses. Consider the potential after Opt adjusted its tree, Φ''. Then the potential change due to Opt's adjustment is

$$\Delta\Phi_2 = \Phi'' - \Phi' = \log B'' - \log B' = \log \frac{B''}{B'} \tag{17}$$

Let Opt access an item z at s_q from level k', raising it to level $k' - 1$ by swapping it with its parent z_1 at s_{q_1}. We denote the t-th parent of z by z_t and the node containing it by s_{q_t}. For all nodes with $s_i.\text{dep} = k' - 1$, except s_{q_1}, $B''_i \leq B_i + 1$ holds, as z_1 goes to level k' from $k' - 1$. For s_{q_1}, $B''_{q_1} \leq B'_q + 2^{k'}$ holds as all the items in layer k' may become bad w.r.t. z. All B''_i with $s_i.\text{dep} < k' - 1$ remain unchanged, i.e., $B''_i = B'_i$ Also all B''_i with $s_i.\text{dep} \geq k'$ remains unchanged except B''_q. Since z_1 from level $k' - 1$ becomes $s_q.guest$ at level k', the bad pairs, if any, associated

to z_1 from level k' becomes good. Hence, $B''_q \le B'_{q_1}$. Taking all together, we have,

$$\begin{aligned}
&B''_i = B'_i \quad \text{(for all } i\text{, s.t., } s_i.\text{dep} < k'-1\text{)}\\
&B''_i = B'_i \quad \text{(for all } i\text{, s.t., } s_i.\text{dep} \ge k'\text{, } i \ne q\text{)}\\
&B''_q \le B'_{q_1}\\
&B''_i \le B_i + 1 \quad \text{(for all } i\text{, s.t. } s_i.\text{dep} = k'-1\text{, } i \ne q_1\text{)}\\
&B''_{q_1} \le B'_q + 2^{k'}
\end{aligned} \tag{18}$$

If Opt brings z to level $k'-p$ ($s_{q_{k'-p}}$ contain z), by performing p swaps, then by similar arguments, we have:

$$\begin{aligned}
&B''_i = B'_i \quad \text{(for all } i\text{, s.t., } s_i.\text{dep} < k'-p\text{)}\\
&B''_i = B'_i \quad \text{(for all } i\text{, s.t., } s_i.\text{dep} \ge k'\text{, } i \ne q\text{)}\\
&B''_q \le B'_{q_1}\\
&B''_i \le B'_i + 1 \quad \text{(for all } i\text{, s.t. } k'-p \le s_i.\text{dep} \le k'-1\text{, } i \ne q_1, q_2, \cdots, q_{k'-p}\text{)}\\
&B''_{q_{k'-p}} \le B'_q + 2^{k'} + 2^{k'-1} + \cdots + 2^{k'-(p-1)}\\
&B''_{q_j} \le B'_{q_{j+1}} \quad \text{(for } j = 1, 2, \cdots, k'-(p+1)\text{)}
\end{aligned} \tag{19}$$

Computation of B'' is shown below:

$$\begin{aligned}
B'' &= \prod_{i=1}^{n} B''_i\\
&\le \left(\prod_{s_i.\text{dep}<k'-p} B'_i\right)\left(\prod_{\substack{k'-p\le s_i.\text{dep}<k'\\ i\ne q_1,q_2,\cdots,q_{k'-p}}} (B'_i+1)\right)\left(\prod_{\substack{s_i.\text{dep}\ge k'\\ i\ne q}} B'_i\right) B''_q B''_{q_1} B''_{q_2}\cdots B''_{q_{k'-(p-1)}} B''_{q_{k'-p}}\\
&\le \left(\prod_{s_i.\text{dep}<k'-p} B'_i\right) 2^{k'}\left(\prod_{\substack{k'-p\le s_i.\text{dep}<k'\\ i\ne q_1,q_2,\cdots,q_{k'-p}}} B'_i\right)\left(\prod_{s_i.\text{dep}\ge k'} B'_i\right) B'_{q_1} B'_{q_2}\cdots B'_{q_{k'-p}}\left(B'_q + 2^{k'} + \cdots + 2^{k'-(p-1)}\right)\\
&\le \left(\prod_{s_i.\text{dep}<k'-p} B'_i\right) 2^{k'}\left(\prod_{\substack{k'-p\le s_i.\text{dep}<k'\\ i\ne q_1,q_2,\cdots,q_{k'-p}}} B'_i\right)\left(\prod_{s_i.\text{dep}\ge k'} B'_i\right) B'_{q_1} B'_{q_2}\cdots B'_{q_{k'-p}} B'_q\left(2^{k'} + \cdots + 2^{k'-(p-1)}\right)\\
&\le B' 2^{k'}\left(2^{k'} + 2^{k'-1} + ... + 2^{k'-p+1}\right)\\
&\le B' 2^{k'} 2^{k'+1}
\end{aligned} \tag{20}$$

Now we compute the potential change due to Opt's swaps:

$$\Delta\Phi_2 = \log\frac{B''}{B'} \le \log 2^{k'} 2^{k'+1} = 2k'+1 \tag{21}$$

OPT may replace several items between two accesses. Let Opt replace w items between the t-th

and the $t+1$-th access. For the r-th replacement, the potential change is less or equal to $(2k'_r+1)$ where as OPT pays (k'_r+p_r). Putting it all together, we get

$$\text{amortized}^{(t)}(\text{ON}) \leq \text{cost}^{(t)}(\text{ON}) + \Delta\Phi_1 + \Delta\Phi_2 \leq k + (2j-k) + \sum_{r=1}^{w}(2k'_r+1) \tag{22}$$
$$\leq 2\left(j + \sum_{r=1}^{w}(k'_r+p_r)\right) \leq 2\text{cost}^{(t)}(\text{OPT})$$

and finally

$$\text{cost}(\text{ON}) = \sum^{t}\text{cost}^{(t)}(\text{ON}) = \sum^{t}\text{amortized}^{(t)}(\text{ON}) - (\Phi^{(t)} - \Phi^{(0)}) \tag{23}$$
$$\leq \sum^{t}\text{amortized}^{(t)}(\text{ON}) \leq \sum^{t} 2\text{cost}^{(t)}(\text{OPT}) = 2\text{cost}(\text{OPT})$$

Now consider the case $j \geq k$. It is easy to observe that $B_r \geq 1$ in this case. Using this in Equation 16, we get $\Delta\Phi_1 \leq j$. Accordingly from Equation 22,

$$\text{amortized}^{(t)}(\text{ON}) \leq \text{cost}^{(t)}(\text{ON}) + \Delta\Phi_1 + \Delta\Phi_2 \leq k + j + \sum_{r=1}^{w}(2k'_r+1) \tag{24}$$
$$\leq 2j + \sum_{r=1}^{w}(2k'_r+p_r) \tag{25}$$
$$\leq 2\left(j + \sum_{r=1}^{w}(k'_r+p_r)\right) \leq 2\text{cost}^{(t)}(\text{OPT})$$

Finally,

$$\text{cost}(\text{ON}) = \sum^{t}\text{cost}^{(t)}(\text{ON}) = \sum^{t}\text{amortized}^{(t)}(\text{ON}) - (\Phi^{(t)} - \Phi^{(0)}) \tag{26}$$
$$\leq \sum^{t}\text{amortized}^{(t)}(\text{ON}) \leq \sum^{t} 2\text{cost}^{(t)}(\text{OPT}) = 2\text{cost}(\text{OPT})$$

The case $\beta > 0$: Let us consider an algorithm ON(β) that maintains an MRU(β) tree for some β. For each request v that ON(β) needs to serve, if $v.\text{dep} = k$ is an MRU tree, then ON(β) needs to pay in the worst case $k+\beta$ (while ON(0) will pay k). According to ON(β), the item of rank 1 is always at depth 0 and the item of rank 2 is always at level 1. For every level $k \geq 2$, we have, $k+\beta \leq (1+\lceil\frac{\beta}{2}\rceil)k$. For the special case of $k=1$, the item with rank 3 can also be at most at depth 2, so the formula holds. Overall we have:

$$\text{cost}(\text{ON}(\beta)) \leq (1+\lceil\frac{\beta}{2}\rceil)\text{cost}(\text{ON}(0)) \leq 2(1+\lceil\frac{\beta}{2}\rceil)\text{cost}(\text{OPT}) \tag{27}$$

Hence ON is $2(1+\lceil\frac{\beta}{2}\rceil)$ access-competitive.

D.2 Proof of Corollary 6

$$\begin{aligned}
\sum_{i=0}^{w} \binom{w}{i} (\frac{w-1}{w})^{w-i} (\frac{1}{w})^i i &= \sum_{i=1}^{w} \binom{w}{i} (\frac{w-1}{w})^{w-i} (\frac{1}{w})^i i \\
&= \sum_{i=1}^{w} \frac{w!}{i!\ (w-i)!} (\frac{w-1}{w})^{w-i} (\frac{1}{w})^i i \\
&= \sum_{i=1}^{w} \frac{(w-1)!}{(i-1)!\ (w-i)!} (\frac{w-1}{w})^{w-i} (\frac{1}{w})^{i-1} \\
&= \sum_{\eta=0}^{w-1} \frac{(w-1)!}{\eta!\ (w-1-\eta)!} (\frac{w-1}{w})^{w-1-\eta} (\frac{1}{w})^{\eta} \qquad \text{(putting } i = \eta + 1) \\
&= \left(\frac{1}{w} + \frac{w-1}{w}\right)^{w-1} \\
&= 1
\end{aligned}$$

D.3 Stochastic Domination

Known results on stochastic domination (see cf. [20]):

Definition 9 (Stochastic Domination) *Let* $\mathtt{X}$ *and* $\mathtt{Y}$ *be two random variables, not necessarily on the same probability space. The random variable* $\mathtt{X}$ *is* stochastically smaller than $\mathtt{Y}$*, denoted by* $\mathtt{X} \preceq \mathtt{Y}$*, if* $\mathbb{P}[\mathtt{X} > z] \leq \mathbb{P}[\mathtt{Y} > z]$ *for every* $z \in \mathbb{R}$*. If additionally* $\mathbb{P}[\mathtt{X} > z] < \mathbb{P}[\mathtt{Y} > z]$ *for some* z*, then* $\mathtt{X}$ *is* stochastically strictly less than $\mathtt{Y}$*, denoted by* $\mathtt{X} \prec \mathtt{Y}$*.*

Theorem 9 (Stochastic Order) *Let* $\mathtt{X}$ *and* $\mathtt{Y}$ *be two random variables, not necessarily on the same probability space.*

1. *Suppose* $\mathtt{X} \prec \mathtt{Y}$*. Then* $\mathbb{E}[U(\mathtt{X})] < \mathbb{E}[U(\mathtt{Y})]$ *for any strictly increasing function* U*.*
2. *Suppose* $\mathtt{X}_1 \prec \mathtt{Y}_1$ *and* $\mathtt{X}_2 \prec \mathtt{Y}_2$*, for four random variables* $\mathtt{X}_1, \mathtt{Y}_1, \mathtt{X}_2$ *and* $\mathtt{Y}_2$*. Then* $a\mathtt{X}_1 + b\mathtt{Y}_1 \prec a\mathtt{X}_2 + b\mathtt{Y}_2$ *for any two constants* $a, b > 0$*.*
3. *Suppose* U *is a non-decreasing function and* $\mathtt{X} \prec \mathtt{Y}$ *then* $U(\mathtt{X}) \prec U(\mathtt{Y})$*.*
4. *Given that* $\mathtt{X}$ *and* $\mathtt{Y}$ *follow the binomial distribution, i.e.,* $\mathtt{X} \sim Bn(n_1, p_1)$ *and* $\mathtt{Y} \sim Bn(n_2, p_2)$*, then* $\mathtt{X} \preceq \mathtt{Y}$ *if and only if the following two conditions holds:* $(1-p_1)^{n_1} \geq (1-p_2)^{n_2}$ *and* $n_1 \leq n_2$*.*